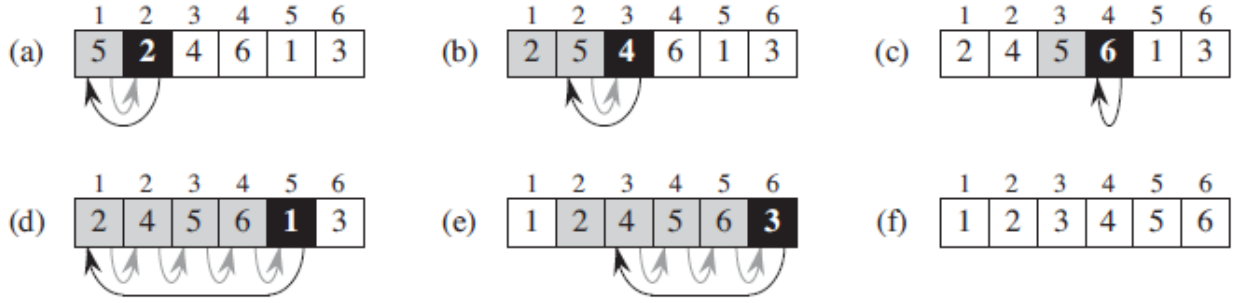
Sorting Algorithms:

Bubble Sort: Bubble sort, also referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

```java
public class BubbleSort {

    public static void main(String[] args) {
        int arr[] = { 3, 60, 35, 2, 45, 320, 5 };

        int n = arr.length;
        int temp = 0;

        for (int i = 0; i < n; i++) {
            for (int j = 1; j < n - i; j++) {
                if (arr[j - 1] > arr[j]) {
                    temp = arr[j - 1];
                    arr[j - 1] = arr[j];
                    arr[j] = temp;
                }
            }
        }

        Arrays.stream(arr).forEach(System.out::println); //2 3  5 35 45 60  320
```

Insertion Sort: Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.



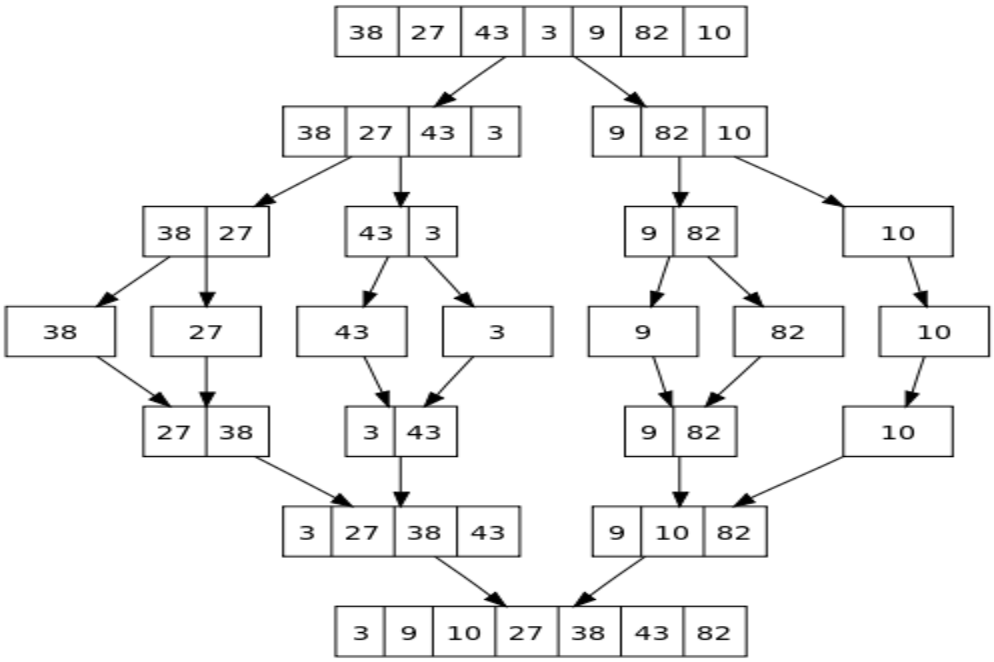
```java
public class MyInsertionSort {

    public static void main(String a[]){
        int[] arr1 = {10,34,2,56,7,67,88,42};
        int[] arr2 = doInsertionSort(arr1);
        for(int i:arr2){
            System.out.print(i);
            System.out.print(", ");
        }
    }

    public static int[] doInsertionSort(int[] input){

        int temp;
        for (int i = 1; i < input.length; i++) {
            for(int j = i ; j > 0 ; j--){
                if(input[j] < input[j-1]){
                    temp = input[j];
                    input[j] = input[j-1];
                    input[j-1] = temp;
                }
            }
        }
        return input;
    }
}
```

Merge Sort: Merge sort is a divide and conquer algorithm.

Steps to implement Merge Sort:

1) Divide the unsorted array into n partitions, each partition contains 1 element. Here the one element is considered as sorted.

2) Repeatedly merge partitioned units to produce new sublists until there is only 1 sublist remaining. This will be the sorted list at the end.



Selection Sort: The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

```
public class MySelectionSort {

    public static int[] doSelectionSort(int[] arr){

        for (int i = 0; i < arr.length - 1; i++)
        {
            int index = i;
            for (int j = i + 1; j < arr.length; j++)
                if (arr[j] < arr[index])
                    index = j;

            int smallerNumber = arr[index];
            arr[index] = arr[i];
            arr[i] = smallerNumber;
        }
        return arr;
    }

    public static void main(String a[]){

        int[] arr1 = {10,34,2,56,7,67,88,42};
        int[] arr2 = doSelectionSort(arr1);
        for(int i:arr2){
            System.out.print(i);
            System.out.print(", ");
        }
    }
}
```

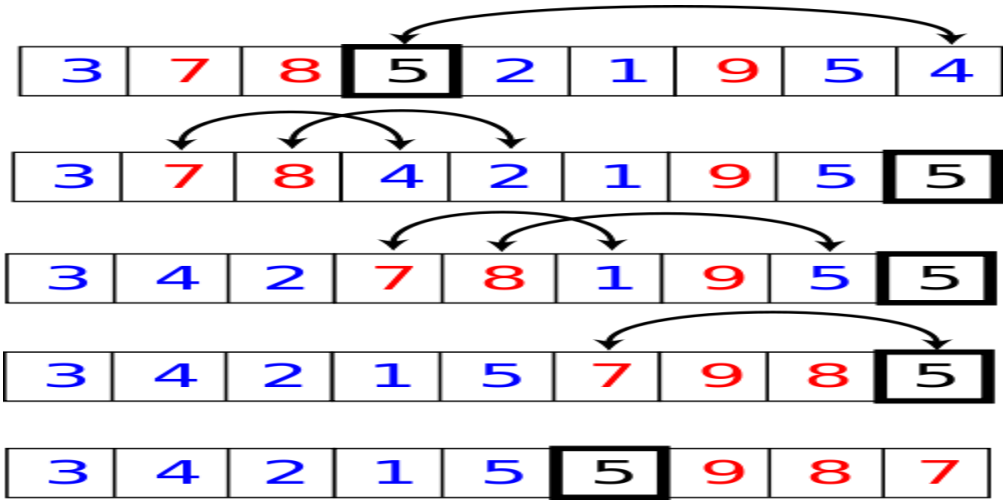
Quick Sort: Quicksort or partition-exchange sort, is a fast sorting algorithm, which is using divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

Steps to implement Quick sort:

1) Choose an element, called pivot, from the list. Generally pivot can be the middle index element.

2) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.



Searching Algorithms:

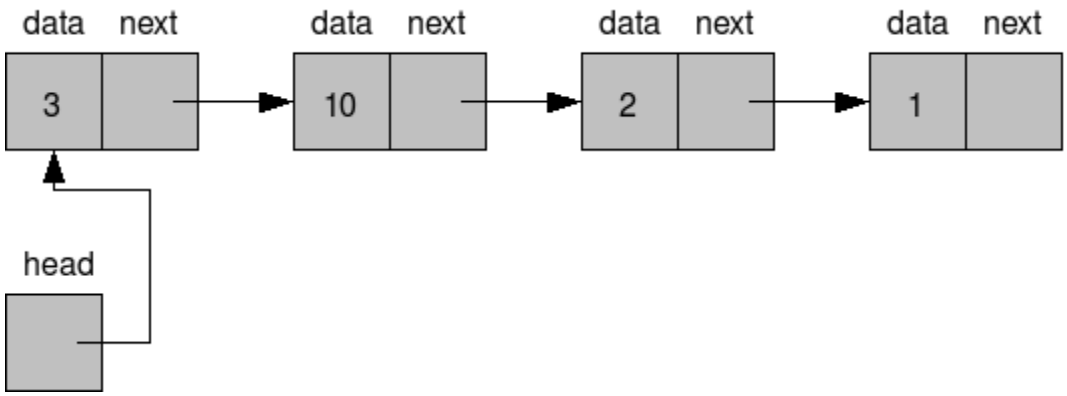
Linear/Sequential Search: Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Binary Search: A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array. In each step, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

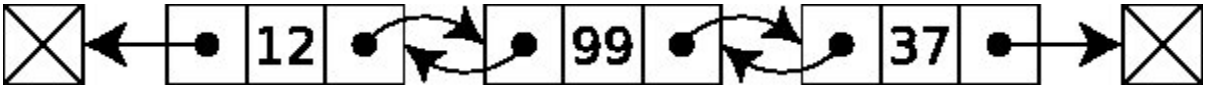
```java
public class BinarySearch {
    public static void main(String[] args) {
        int arr[] = { 2, 3, 4, 10, 40 };
        int searchElement = 10;
        int left = 0;
        int right = arr.length - 1;

        int result = binarySearch(arr, left, right, searchElement);

        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index " + result);
    }

    static int binarySearch(int arr[], int left, int right, int x) {
        if (right >= left) {
            int mid = left + (right - left) / 2;
            System.out.println("mid " +mid);
            // If the element is present at the
            // middle itself
            if (arr[mid] == x)
                return mid;

            // If element is smaller than mid, then
            // it can only be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, left, mid - 1, x);

            // Else the element can only be present
            // in right subarray
            return binarySearch(arr, mid + 1, right, x);
        }

        // We reach here when element is not present
        // in array
        return -1;
    }
```

Data Structures:

Single Linked List: Singly Linked Lists are a type of data structure. It is a type of list. In a singly linked list each node in the list stores the contents of the node and a pointer or reference to the next node in the list. It does not store any pointer or reference to the previous node. It is called a singly linked list because each node only has a single link to another node. To store a single linked list, you only need to store a reference or pointer to the first node in that list. The last node has a pointer to nothingness to indicate that it is the last node.



Double Linked List: A doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient, because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Stack: A Stack is an abstract data type or collection where in Push, the addition of data elements to the collection, and Pop, the removal of data elements from the collection, the major operations are performed on the collection. The Push and Pop operations are performed only at one end of the Stack which is referred to as the 'top of the stack'.

In other words, a Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

Queue: A queue is a kind of abstract data type or collection in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear terminal position, called as enqueue, and removal of entities from the front terminal position, called as dequeue. The queue is called as First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.